Basic Python Programming

CS 3753 Data Science

Prof. Weining Zhang

Topics

1. Whitespace Formatting
2. Import Modules
3. Variable and Arithmetic
4. Functions
5. If-then
6. Loop
7. Built-in Collection Types

Variables and Arithmetic

The principal built-in types in Python are

- numerics, sequences, mappings, classes, instances and exceptions.

In Python, variables do not need type declaration.

Python provides a set of tools for processing numeric objects.

- Expression operators
  - +, -, *, /, >>, **, &, etc.
- Built-in mathematical functions
  - pow, abs, round, int, hex, bin, etc.
- Utility modules
  - math: pi, e, ceil(), floor(), sqrt(), sin(), cos(), log(), log10(), etc.
  - random: random(), randrange(), uniform(), choices(), etc.

In [ ]:
```
a = 11.0
b = 3
c = 5
d = True

print(a*2, ", ", b/c, ", ", b//c, ", ", a%b, ", ", (d & False))
print(divmod(a,b), ", ", pow(b, 2), ", ", a**2)

import math, random
print(math.sqrt(math.pi*a), ", ", math.log2(math.pow(a, c)), ", ", r
andom.choices([a, b, c, d]))
```

Functions

- A function is a rule for taking zero or more inputs and returning a corresponding output. In Python, we typically define functions using def.
- Python functions are first-class, which means that we can assign them to variables and pass them into functions just like any other arguments
- Python provides many built-in functions (see here for documentation (https://docs.python.org/3/library/functions.html))

def <functionName> (<parameters>) :
    <statements>

def <functionName> (<parameters>) :
    <statements>

Strings

- Can be 'single quote' or "double quote" but the quotes must match
- Escape for special characters: \t, \n, \"
- Raw strings r"", e.g., r"\t" vs "\t"
- Multiline strings using """ ... """
- Python string functions, such as, format(), parse(), etc.
  - documentation and examples (https://docs.python.org/3/library/string.html)

In [ ]:

```
single_quoted_string = 'data science'
double_quoted_string = "data science"

tab_string = "\t" # represents the tab character
len(tab_string) # is 1
print(tab_string, len(tab_string))

not_tab_string = r"\t" # represents the characters '\' and 't'
len(not_tab_string) # is 2
print(not_tab_string, len(not_tab_string))

multi_line_string = """This is the first line.
    and this is the second line
    and this is the third line"""
print(multi_line_string)

print("formated output: a={0:3.2f} and b={1:5d}".format(a, b))
```

Built-in Collection Types

Provided by the collections module

- List: [e1, e2, ..., ek]
- Tuple: (e1, e2, ..., ek) or e1, e2, ..., ek
- Set: {e1, e2, ..., ek}
- Dictionary: {k1:v1, k2:v2, ..., kk:vk}

In [ ]:
```
import collections
```

List

- A list is an ordered collection of objects, can mix different type of objects in one list, and can have list nested in a list
- List is a class with many attributes and functions
- Elements in a list can be accessed by indexing and list comprehension (a loop type statement that defines a complex access pattern)

In [ ]:
```
L = [-17.5, "kilo", 49, "V", ["ram", 5, "echo"], 7]
L
```

Access Individual List Elements

- position index starts at 0
- can access element using both forward (positive) index and backward (negative) index

In [ ]:
```
a = "L[1][0] = {0}, L[-5][0] = {1}"
print("len(L)={}".format(len(L)))
print("L[0] = {0}, L[-6] = {1}".format(L[0], L[-6]))
print("L[1][0] = {0}, L[-5][0] = {1}".format(L[1][0], L[-5][0]))
print("L[4][2] = {0}, L[-2][-1] = {1}".format(L[4][2], L[-2][-1]))
```

List is a class with many built-in attributes and functions

In [ ]:
```
x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
x
```

In [ ]:
```
dir(x)
```

In [ ]:
```
help(x.insert)
#x.insert(4, 'a')
#x.remove('a')
x
```

Select a range of elements in a list

- use range index: list-name[start : end : increment]

In [ ]:
```
x[:]
```

In [ ]:
```
x[1::2]
```

In [ ]:
```
[0] * len(x[1::2])
```

In [ ]:
```
x[1::2] = [0] * len(x[1::2]) # setting odd position to 0
x
```

List Comprehension

- Provides a way to transform a list into another list, by choosing only certain elements, or by transforming elements, or both.
- Use (nested) for loop syntax

In [ ]:
```
even_numbers = [x for x in range(5) if x % 2 == 0] # [0, 2, 4]
squares = [x * x for x in range(5)] # [0, 1, 4, 9, 16]
even_squares = [x * x for x in even_numbers] # [0, 4, 16]
```

In [ ]:
```
leaps = [y for y in range(1900, 1940) \
         if (y % 4 == 0 and y % 100 != 0) or (y % 400 == 0)]
leaps
```

In [ ]:
```
leaps[:]
```

In [ ]:
```
print("size of leaps = {0}, type of leaps is: {1}".format(len(leaps)
,
      type(leaps)))
```

Apply Python's built-in (global) functions on list

In [ ]:
```
x = [-2, 9, 7, -4, 3]
x
```

In [ ]:
```
print(all(x), any(x), len(x), min(x), max(x), sum(x))
```

Combine or extend lists

- Can use list functions, such as append(), extend()
- can also use operators, such as +, *

In [ ]:
```
print(x)
x.append(0)
x
```

In [ ]:
```
#x.extend([10, 20, 30])
x.append([10, 20, 30])
x
```

In [ ]:
```
print(x * 2)
```

In [ ]:
```
x + [15, 25, 35]
```

Use of the range function

A range function range() generates a sequence integers

- range(end)
- range(start, end)
- range(start, end, increment)

In [ ]:
```
print(list(range(5)), list(range(9, 14)), tuple(range(10, -11, -5)))
```

Use iterators

- An iterator is an object that can move through a list-like collection, one element at a time.
- It must be assigned with a list-like collection
- Use next() to access the elements

In [ ]:
```
product = 1
i = iter([1, 2, 4, 8])
while True:
    try:
        product *= next(i)
        print(product)
    except StopIteration:
        break
```

Tuple

Tuple is a list with a fixed size. In other words, a tuple is immutable.

- You can't add elements to a tuple. Tuples have no append or extend method.

In [ ]:
```
hair = "black", "brown", "blonde", "red"
eyes = ("brown", "hazel", "amber", "green", "blue", "gray")
colors = (hair, eyes)
colors
```

In [ ]:
```
colors[1][3:-1]
```

In [ ]:
```
things = (1, -7.5, ("pea", (5, "Xyz"), "queue"))
things[2][1][1][2]
```

Named Tuple

A tuple type where a name is associated with the structure

In [ ]:
```
Sale = collections.namedtuple("Sale",
    "productid customerid date quantity price")
sales = []
sales.append(Sale(432, 921, "2008-09-14", 3, 7.99))
sales.append(Sale(419, 874, "2008-09-15", 1, 18.49))
sales
```

In [ ]:
```
total = 0
for sale in sales:
    print(sale) # print the tuple
    print(sale[3], sale.price) # print quantity and price
    total += sale.quantity * sale.price
total # $42.46
```

Set

The collection representing the standard set concept, with operators for

- union, intersection, difference, in, etc.

In [ ]:
```
S1 = {7, "veil", 0, -29, ("x", 11), "sun", frozenset({8, 4, 7}), 913
}
S2 = {"pecan", "pie", 7, "sun"}
print("S1 = {0}\nS2 = {1}".format(S1, S2))
len(S1)
```

In [ ]:
```
s3= S1 | S2 # union
s4 = S1.union(S2)
print(s3)
print(s4)
```

In [ ]:
```
S1 & S2 # intersect
```

In [ ]:
```
S1 - S2 # set difference
```

In [ ]:
```
S1 ^ S2 # symmetric difference
```

In [ ]:
```
print("S1 = {0}\nS2 = {1}".format(S1, S2))
```

Dictionary

- Also called the map, which maps keys to correspondant values (where a value can be any type of object, such as list, tuple, set, and dictionary)
- A dict has functions, such as keys(), values(), items(), etc.
- A value can be accessed by key, such as, dict-name[key]

In [ ]:
```
d = {"root": 18, "blue": [75, "R", 2], 21: "venus", -14: None,
"mars": "rover", (4, 11): 18, 0: 45}
d
```

In [ ]:
```
d.keys()
```

In [ ]:
```
d.values()
```

In [ ]:
```
d.items()
```

In [ ]:
```
for key, value in d.items():
    print("key: {0}, value: {1}".format(key, value))
```

In [ ]:
```
for k in d:
    print("k: {0}, d[k]: {1}".format(k, d[k]))
```

Exercise

Use a list comprehension to build a dict from the following list of tuples.

L = [("name", "John"), ("age", 23), ("salary", "35k"), ("phone", "210-458-5757")]

In [ ]:
```
L = [("name", "John"), ("age", 23), ("salary", "35k"), ("phone", "21
0-458-5757")]

d3 = {t[0]: t[1] for t in L }

d3
```

Exercise

Use indexing to get the character 'R' from the following dict.

d5 = {"root": 18, "blue": [75, "Rick", 2], 21: "venus", -14: None, "mars": "rover", (4, 11): 18, 0: 45}

In [ ]:
```
d5 = {"root": 18, "blue": [75, "Rick", 2], 21: "venus", -14: None,
"mars": "rover", (4, 11): 18, 0: 45}

d5["blue"][1][0]
```

Exercise

Write a fragment of code to create a list of tuples from the following dict, so that each tuple contains the name, age, dept, and salary of one person.

d4 = {"name": ["adams", "john", "steve", "linda"],
"age": [20, 43, 19, 25],
"dept": ["IT", "Sales", "IT", "HR"],
"salary": [35000, 51000, 36000, 40000]}

In [ ]:
```
d4 = {"name": ["adams", "john", "steve", "linda"],
         "age": [20, 43, 19, 25],
         "dept": ["IT", "Sales", "IT", "HR"],
         "salary": [35000, 51000, 36000, 40000]}
[tuple(d4[k][x] for k in d4.keys()) for x in range(4)]
```

Python Script

- Store in scriptFileName.py
- Contains variables, functions, constants
- Can run by

  python scriptFileName.py
- or from Jupyter notebook using %scriptFileName.py or !scriptFileName.py

In [ ]:
```
# %load quadratic.py
%load quadratic.py
```

In [ ]:
```
%run quadratic.py
```

Object Programming

- Define class

  class className :
```
def __init__(self, otherParameters) :
   statements

def otherFunction( parameters ) :
   statements
```

In [ ]:
```
class Name:
    def __init__(self, f, l):
        self.first = f
        self.last = l
    def getFirst(self):
        return self.f
    def getLast(self):
        return self.last
    def toString(self):
        return "name: " + self.first + " " + self.last
    def display(self):
        print(self.toString())

class Person:
    def __init__(self, n, a):
        self.name = n
        self.age = a
    def getName(self):
        return self.name
    def getAge(self):
        return self.age
    def toString(self):
        return self.name.toString() + "\nAge: " + str(self.age)
    def display(self):
        print(self.toString())
```

In [ ]:
```
a = Name("Steve", "Goodman")
a.display()
```

In [ ]:
```
p = Person(a, 25)
p.display()
```